CHAPTER 3: CONTROL FLOW

The control flow statements of a language specify the order in which things get done. We have already met the most common control flow constructions of C in earlier examples; here we will complete the set, and be more precise about the ones discussed before.

3.1 Braces

The braces { and } are used to group statements together so that they are syntactically equivalent to a single statement. The braces that surround the statements of a function are one obvious example; braces around multiple statements after an if or else or while or for are the other. Braces can also be used to delimit a "block" in which local variables can be declared; we will talk about this in Chapter 4.

3.2 If-Else

if-else is used to make decisions. Formally, the syntax is either

```
if (expression)
        statement
```

or

```
if (expression)
        statement
else
        statement
```

The *expression* is evaluated; if it is true (that is, if the *expression* has a non-zero value), the first statement is done. If it is false (zero) ("false") and if there is an else part, the second statement is done.

Since an if simply tests the numeric value of an expression, certain coding shortcuts are possible. The most obvious is writing

```
if (expression)
```

instead of

```
if (expression != 0)
```

Sometimes this is natural and clear; at other times it is cryptic and unsafe, so watch out.

Because the **else** part of an **if-else** is optional, there is an ambiguity when an **else** is omitted from a nested **if** sequence. This is resolved in the usual way — the **else** is associated with the previous un-**else**'ed **if**. For example, in

```
if (n > 0)
        if (a > b)
                z = a;
        else
                z = b;
```

the **else** goes with the inner **if**, as we have shown by indentation. If that isn't what you want, braces must be used to force the proper association:

```
if (n > 0) {
        if (a > b)
                z = a;
}
else
        z = b;
```

The ambiguity is especially pernicious in situations like:

```
if (n > 0)
        for (i = 0; i < n; i++)
                if (s[i] > 0) {
                        printf("...");
                        return(i);
                }
else
        printf("error — n is zero\n");
```

The indentation shows unequivocally what you want, but the compiler doesn't get the message. This kind of bug can be very hard to find.

By the way, notice that there is a semicolon after **z = a** in

```
if (a > b)
        z = a;
else
        z = b;
```

This is because grammatically, a *statement* follows the **if**, and statements must be terminated by a semicolon.

3.3 Conditional Expressions

This last example of course computes in z the maximum of a and b. The ternary operator ? : provides an alternate way to write this and similar constructions:

```
        z = (a > b) ? a : b;        /* z = max(a, b) */
```

In the expression

```
        p ? q : r
```

the expression p is evaluated. If it is non-zero (true), then the value of the conditional expression is the value of the expression q; otherwise it is r. Only one of q and r is evaluated. If q and r are of different types, the type of the result is based on the conversion rules discussed in the previous chapter. For example, if f is a float, and n is an int, then the expression

```
  (n > 0) ? f : n
```

is of type float regardless of whether n is positive or not.

Parentheses are not necessary around the first expression of a conditional expression, since the precedence of ? : is very low. They are advisable anyway, however, since they make the condition part of the expression easier to see.

The conditional expression often leads to succinct code. For example, this loop prints N elements of an array, 10 per line, separated by ~~three~~ blanks, with no extra characters, and each line (even the last) properly terminated by a single newline.

3 blanks ?

```
  for (i = 0; i < N; i++)
          printf("%d%s", a[i], (i==N-1 || i%10==9) ? "\n" : "   ");
```

A newline is printed every 10 elements, and after the Nth. All other elements are followed by three blanks. Although this might look tricky, it's instructive to try to write it without the conditional expression.

Exercise 3-1: Write the printing loop without a conditional expression. □

Exercise 3-2: Rewrite the function lower (see Chapter 2) to convert upper case letters to lower case, using a conditional expression instead of if-else. □

3.4 Else-If

The construction

```
if (expression)
        statement
else if (expression)
        statement
else if (expression)
        statement
else
        statement
```

occurs so often in programming that it is worth a brief separate discussion. This sequence of if's is the most general way of writing a multi-way decision. The *expression*'s are evaluated in order; if any *expression* is true, the *statement* associated with it is executed, and then the whole chain is exited. The code for each *statement* is either a single statement, or a group in braces.

The last **else** part handles the "none of the above" or default case where none of the other conditions was satisfied. Sometimes there is no explicit action for the default; in that case it can be omitted.

To illustrate a three-way decision, here is a binary search function that decides if a particular value x occurs in the sorted array v. It returns the position (a number between 0 and n−1) if x occurs in v, and −1 if not.

```
binary(x, v, n)         /* find x in v[0] ... v[n-1] */
int x, v[ ], n;
{
        int low, high, mid;

        low = 0;
        high = n - 1;
        while (low <= high) {
                mid = (low+high) / 2;
                if (x == v[mid])
                        return(mid);
                else if (x < v[mid])
                        high = mid - 1;
                else
                        low = mid + 1;
        }
        return(-1);
}
```

The fundamental decision is whether x is less than, equal to, or greater than v[mid] at each step; this is a natural for **else-if**. Since the first case ends with a **return**, there is no need for the **else** that follows; we could remove it. But the version presented is preferable, since it better shows the three-way nature of the decision.

3.5 Switch

The **switch** statement is a special multi-way decision maker that tests whether an expression matches one of a number of *constant* values, and branches accordingly. In Chapter 1 we wrote a program to count the number of each digit, white space, and all other characters, using an **if** ... **else if** ... **else**. Here is the same program with **switch**.

```
main()        /* count digits, white space, others */
{
        int c, i, nwhite, nother, ndigit[10];

        nwhite = nother = 0;
        for (i = 0; i < 10; i++)
                ndigit[i] = 0;

        while ((c = getchar()) != EOF)
                switch (c) {
                        case '0':
                        case '1':
                        case '2':
                        case '3':
                        case '4':
                        case '5':
                        case '6':
                        case '7':
                        case '8':
                        case '9':
                                ndigit[c-'0']++;
                                break;
                        case ' ':
                        case '\n':
                        case '\t':
                                nwhite++;
                                break;
                        default:
                                nother++;
                                break;
                }

        printf("digits = ");
        for (i = 0; i < 10; i++)
                printf("%d ", ndigit[i]);
        printf("\nwhite space = %d, other = %d\n", nwhite, nother);
}
```

The **switch** compares the expression in parentheses (in this program the character **c**) to all the cases. Each case must be labelled by an *integer constant*, which includes character constants like '0'. If a case matches, execution starts at that case. The case labelled **default** is executed if none of the other cases is satisfied. (A **default** is optional; if it isn't there, and none of the cases matches, no action at all takes place.) Cases and default can occur in any order.

The **break** statement causes an immediate exit from the **switch**. Because cases are just labels, after the code for one case is done, execution *falls through* to the next unless you take explicit action to escape. **break** and **return** are the most common ways to leave a **switch**. **break** also causes an immediate exit from **while** and **for** loops as well, as will be discussed later in this chapter.

Falling through cases is a mixed blessing. On the positive side, it allows multiple cases for a single action, as with the blank, tab or newline in this example. But it also implies that normally each case must end with a **break** to prevent falling through to the next. Falling through from one case to another is not robust, being prone to disintegration when the program is modified. With the exception of multiple labels for a single computation, fall-throughs should be used sparingly.

As a matter of good form, put a **break** after the last case (the **default** here) even though it's logically unnecessary. Some day when another case gets added at the end, this bit of defensive programming will save you.

Exercise 3-3: Write a function **expand(s, t)** which converts characters like newline into visible escape sequences like **\n** as it copies **s** to **t**. Use a **switch**. □

3.6 Loops — while and for

We have already encountered the **while** and **for** loops. In

```
while (expression)
        statement
```

the *expression* is evaluated. If it is non-zero, *statement* is executed and *expression* is re-evaluated. This cycle continues until *expression* becomes false, at which point execution resumes after *statement.*

The for statement

```
for (init; expression; re-init)
        statement
```

is equivalent to

```
        init;
        while (expression) {
                statement
                re-init;
        }
```

init and *re-init* are expressions. Any of the three parts can be omitted. If *init* or *re-init* is left out, it is simply dropped from the expansion. If *expression* is not present, it is taken as permanently true, so

```
for ( ; ; ) {
        ...
}
```

is an "infinite" loop, presumably to be broken by other means (such as a break or return).

Whether to use while or for is largely a matter of taste. For example, in

```
/* skip white space characters */
while ((c = getchar( )) == ' ' || c == '\n' || c == '\t)
        ;
```

there is no initialization or re-initialization, so the while seems most natural.

The for is clearly superior when there is a simple initialization and re-initialization, since it keeps the loop control statements close together and visible at the top of the loop. This is obvious in

```
        for (i = 0; i < N; i++)
```

which is the C idiom for processing the first N elements of an array, the analog of the Fortran or PL/I DO loop.

As a larger example (which also makes use of some other constructs we've discussed), here is another version of atoi for converting a string to its numeric equivalent. This one is more general; it copes with optional leading white space and an optional + or − sign. (Chapter 4 shows atof, which does the same conversion for floating point numbers.)

The basic structure of the program reflects the form of the input:

```
skip blanks, if any
get sign, if any
get integer part
```

Each step processes its part if present, and leaves things in a clean state for the next part. If the whole process terminates on a character that is not white space, then there was some sort of error, although we won't do anything about that yet.

```
atoi(s)/* convert s to integer */
char s[ ];
{
        int i, n, sign = 1;

        for (i = 0; s[i] == ' ' || s[i] == '\n' || s[i] == '\t'; i++)
                ;       /* skip white space */
        if (s[i] == '+' || s[i] == '-')       /* sign */
                sign = (s[i++]=='+') ? 1 : -1;
        for (n = 0; s[i] >= '0' && s[i] <= '9'; i++)  /* integer part */
                n = 10 * n + s[i] - '0';
        return(sign * n);
}
```

The advantages of keeping loop control centralized are even more obvious for nested loops. The following function is a Shell sort, for sorting an array of integers. The basic idea of the Shell sort is that in early stages, far apart elements are compared, rather than adjacent ones, as in simple interchange sorts. This tends to eliminate large amounts of disorder quickly, so later stages have less work to do. The interval between compared elements is gradually decreased to one, at which point the sort effectively becomes an adjacent interchange method.

```
shell(v, n)     /* sort v[0] ... v[n-1] into increasing order */
int v[ ], n;
{
        int gap, i, j, k;

        for (gap = n/2; gap > 0; gap /= 2)
                for (i = gap; i < n; i++)
                        for (j=i-gap; j>=0 && v[j]>v[j+gap]; j -= gap) {
                                k = v[j];
                                v[j] = v[j+gap];
                                v[j+gap] = k;
                        }
}
```

The outermost loop controls the gap between compared elements, shrinking it from n/2 by a factor of two each pass until it becomes zero. The middle loop compares elements separated by gap; the innermost loop reverses any that are out of order. Since gap is eventually reduced to one, all elements are eventually ordered correctly.

Writing this code with while expands it by *nine* lines; the conciseness of the for adds clarity.

3.7 Loops — do-while

The while and for share the desirable attribute of testing the loop at the top, rather than at the bottom, as we discussed in Chapter 1. The third loop in C, the do-while, tests at the bottom *after* making one pass through the body. The syntax is

```
do
        statement
while (expression)
```

statement is executed, then *expression* is evaluated. If it is true, *statement* is evaluated again, and so on. If the expression becomes false, the loop terminates.

As might be expected, the do-while is much less used than while and for. Nonetheless, it is from time to time valuable, as in this function for converting a number to a character string, the inverse of atoi. The job is slightly more complicated than might be thought at first, because the easy methods of generating the digits generate them in the wrong order. We have chosen here to generate the string backwards, then reverse it.

```
itoa(n, s)      /* convert n to characters in s */
char s[ ];
int n;
{
        int c, i, j, sign = 1;

        if (n < 0) {  /* record sign */
                sign = -1;
                n = -n;
        }
        i = 0;
        do {            /* generate digits in reverse order */
                s[i++] = n % 10 + '0';
        } while ((n /= 10) > 0);
        if (sign < 0)
                s[i++] = '-';
        s[i] = '\0';
        for (i--, j = 0; j < i; i--, j++) {       /* reverse s */
                c = s[i];
                s[i] = s[j];
                s[j] = c;
        }
}
```

COMMAS!

The do-while is necessary, or at least convenient, since at least one character must be installed in the array s, regardless of the value of n. We also used braces around the single statement that makes up the body of the do-while, even though it is unnecessary, so the hasty reader will not mistake the while part for the *beginning* of a **while** loop.

Exercise 3-4: Our version of **itoa** does not handle "negative infinity", that is, the value of n equal to -2^{wordsize}. Explain why not. Modify it to print that value correctly. □

3.8 Break and Continue

It is useful to be able to control loop exit by other means than simply testing at the top. The **break** statement provides an early exit from **for**, **while**, and **do**, just as from **switch**.

The following program trims blanks and tabs from each line of input, using a **break** to exit from a loop when the last non-blank, non-tab is found.

```
main()      /* remove trailing blanks and tabs */
{
        int n;
        char line[MAXLINE];

        while ((n = getline(line, MAXLINE)) >= 0) {
                while (--n >= 0)
                        if (line[n] != ' ' && line[n] != '\t')
                                break;
                line[n+1] = '\0';
                printf("%s\n", line);
        }
}
```

getline returns the length of the line with the terminating \n removed. The inner **while** loop starts at the last character of **line** (recall that --n decrements n before using the value), and scans backwards looking for a non-blank, non-tab. The loop is broken when one is found, or when n becomes negative (that is, when the beginning of the string is reached). This is correct behavior even when an empty line is encountered, for which n is zero.

An alternative to **break** is to put the testing in the loop itself:

```
while ((n = getline(line, MAXLINE)) >= 0) {
        while (--n >= 0 && (line[n] == ' ' || line[n] == '\t'))
                ;
        ...
}
```

This is inferior to the previous version, because the test is much harder to understand. Compound tests with parentheses and different operators should be avoided.

The **continue** statement is related to **break** (although empirically much less used); it causes the *next iteration* of the enclosing loop (**for**, **while**, **do**) to begin. In the **while** and **do**, this means that the test part is executed immediately; in the **for**, control passes to the re-initialization step. (**continue** applies only to loops, not to switch.)

As an example, this fragment processes only positive elements in the array a; negative values are skipped.

```
for (i = 0; i < N; i++) {
        if (a[i] < 0)  /* skip negative */
                continue;
        ...     /* do positive */
}
```

continue seems most often used in situations like this, when the part of the loop that follows is complicated, so that turning a logical condition around and indenting another level would complicate the program too much.

3.9 goto's and Labels

C provides the infinitely-abusable goto statement, and labels to branch to. Formally, the goto is never necessary, and in practice it is almost always easy to write code without it. Certainly we don't use goto in this book.

Nonetheless, we will suggest a few situations where goto's may find a place. The most common use is in error-handling code when it is necessary to abandon processing in some deeply nested structure.

```
        for ( ... )
                for ( ... )
                        ...
                        if (disaster)
                                goto error;
...

error:
        clean up the mess
```

This organization is handy if the error-handling code is non-trivial, and if errors can occur in several deeply nested places.

As another example, consider the problem of finding the first negative element in a two-dimensional array. One possibility is

```
        for (i = 0; i < N; i++)
                for (j = 0; j < M; j++)
                        if (v[i][j] < 0)
                                goto found;
        /* didn't find */

found:
        /* found one at position i, j */
```

This can be written without a goto, at the price of some repeated tests or an extra variable; this is always the case.

```
found = 0;
for (i = 0; i < N && !found; i++)
        for (j = 0; j < M && !found; j++)
                found = v[i][j] < 0;
if (found)
        /* it was at i-1, j-1 */
else
        /* not found */
```

A label has the same form as a variable name. It can be attached to any statement in the same function as the goto. It is currently possible to branch into a set of statements in braces, or to the else part of an if, etc., but these are bad practices indeed.